FVPP: SMT-rješavači i simbolička verifikacija programa

Verifikacija složenijih logičkih sustava

Pripremio: izv. prof. dr. sc. Alan Jović

Ak. god. 2022./2023.

Sadržaj

- Uvod u SMT-rješavače
- Teorije
- Primjer primjene i veza sa SAT-rješavačima
- Uvod u simboličku verifikaciju programa

UVOD U SMT-RJEŠAVAČE

SMT-rješavači

- **SMT-rješavač (engl. *Satisfiability Modulo Theory, SMT solver*)** je računalni program koji **odlučuje o zadovoljivosti formule predikatne logike prvoga reda** (FOPL) uz dodatak **teorija** kao što su teorija jednakosti, cijelih i realnih brojeva, polja, itd.
  - Teorije koje rješava SMT-rješavač su puno složenije od onih koje rješava SAT-rješavač
- Primjena SMT-rješavača je u raznim područjima znanosti i tehnike, a posebice u:
  - **formalnoj verifikaciji** – automatsko dokazivanje teorema
  - **simboličkom izvršavanju programa** – analiza i verifikacija rada programa

SMT-rješavači

- Problemi koje SMT-rješavači rješavaju mogu imati više stotina pa i tisuća klauzula kao što je ova:

  `p ∨ ∀xP(x) ∨ a=f(b-c) ∨ g(g(b))≠c ∨ (a-c)≤7`

- U programima, klauzule opisuju ograničenja nad varijablama, koja su zadana strukturom programa ili njegovim preduvjetima
- SMT-rješavači se smatraju programima koji učinkovito rješavaju probleme s ograničenjima pa ih se još naziva **rješavačima ograničenja** (engl. *constraint solver*)

SMT-rješavači

- Najčešće samostalni i brzi programi koje pozivaju drugi alati, npr. oni za simboličko izvršavanje programa ili oni koji rješavaju neke praktične probleme (npr. izrada rasporeda sati)
- Suvremeni SMT-rješavači uspješno rade s oko 100 000 Booleovih varijabli (i ekvivalentnim brojem ne-Booleovih varijabli) i s programima od oko 100 000 linija koda
- Primjeri ograničenja na koje SMT-rješavači mogu naići, od kojih neka ne moraju nužno znati riješiti:
  - Linearna ograničenja: $5x + 6 < 100$
  - Nelinearna ograničenja: $x \cdot y + 2 < 35$
  - Neinterpretirane funkcije: $f(x) < 4$
- Zadatak SMT-rješavača je da pronađe zadovoljavajuće pridruživanje vrijednosti varijabli uz zadana ograničenja

TEORIJE

Teorije

- **Teorija = gradivni dio logičke formule koja tvori ograničenje**
- Matematički jasno i precizno opisana logička struktura, zadana je svojim **potpisom** (domenom djelovanja) i **aksiomima** (tvrdnjama koje u njoj vrijede)
- Svaka teorija, kada se razmatra kao **problem odlučivanja**, može biti odlučljiva (*decidable*) ili neodlučljiva (*undecidable*)
- Napomena: iako odlučljive, neke teorije imaju visoku složenost odlučivanja (npr. eksponencijalnu)

Primjer – Teorija jednakosti T_E

- **Potpis** teorije jednakosti T_E :
  - Simboli za neinterpretirane funkcije (npr. f, g, ...), predikate (npr. P, Q, ...), varijable (npr. x, y, z...) i konstante (npr. a, b, ...)
  - Binarni predikat "=", interpretiran sa značenjem koje mu daju aksiomi
- **Aksiomi** teorije jednakosti T_E :
  1. $\forall x.\ \ x = x$ (refleksivnost)
  2. $\forall x,y.\ \ x = y \rightarrow y = x$ (simetrija)
  3. $\forall x,y,z.\ \ x = y \wedge y = z \rightarrow x = z$ (tranzitivnost)
  4. Za svaki pozitivni cijeli broj *n* i *n*-arni funkcijski simbol *f*,

     $\forall x_1,\ldots,x_n, \forall y_1,\ldots,y_n.\ \ x_1 = y_1 \wedge \ldots\ x_n = y_n \rightarrow f(x_1,\ldots,x_n) = f(y_1,\ldots y_n)$

     (kongruentnost funkcija)
  5. Za svaki pozitivni cijeli broj *n* i *n*-arni predikatni simbol *P*,

     $\forall x_1,\ldots,x_n, \forall y_1,\ldots,y_n.\ \ x_1 = y_1 \wedge \ldots\ x_n = y_n \rightarrow P(x_1,\ldots,x_n) = P(y_1,\ldots y_n)$

     (kongruentnost predikata)

Primjer – Teorija jednakosti T_E

Primjeri:

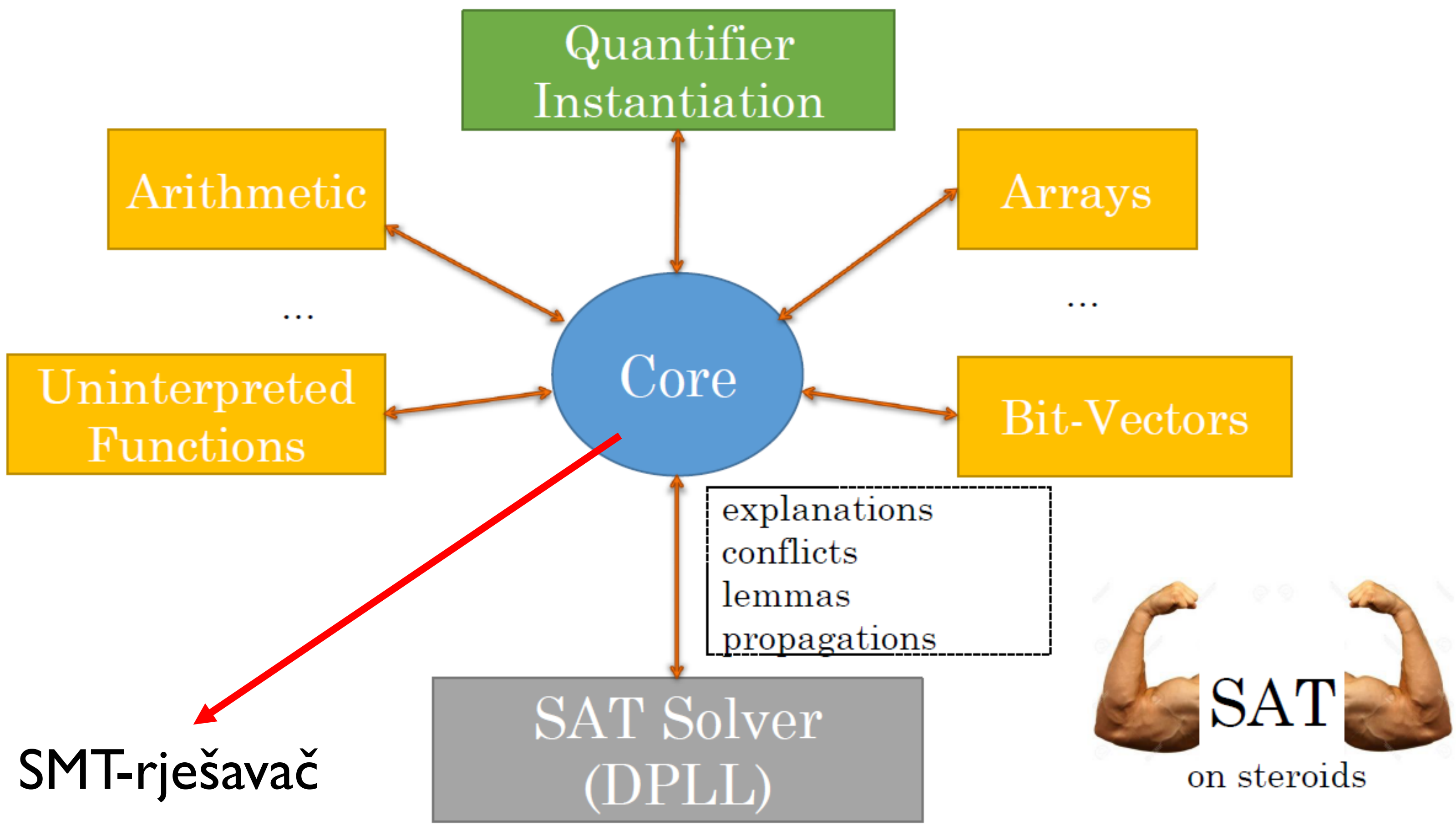
- za binarni funkcijski simbol f i n = 2 vrijedi tvrdnja:

$$\forall x_1, x_2, y_1, y_2 .\ x_1 = y_1 \wedge x_2 = y_2 \rightarrow f(x_1, x_2) = f(y_1, y_2)$$

- vrijedi tvrdnja:

$$a = b \wedge b = c \rightarrow g(f(a), b) = g(f(c), a)$$

SMT-rješavači i podržane teorije



Teorije – odlučljivost

Theory	Description	Full	QFF
T_E	equality	no	yes
T_{PA}	Peano arithmetic	no	no
$T_{\mathbb{N}}$	Presburger arithmetic	yes	yes
$T_{\mathbb{Z}}$	linear integers	yes	yes
$T_{\mathbb{R}}$	reals (with $\cdot$)	yes	yes
$T_{\mathbb{Q}}$	rationals (without $\cdot$)	yes	yes
T_{RDS}	recursive data structures	no	yes
T_{RDS}^{+}	acyclic recursive data structures	yes	yes
T_A	arrays	no	yes
$T_A^{=}$	arrays with extensionality	no	yes

- QFF – formula bez kvantifikatora – fragment teorije – npr. $P(f(a), b) \wedge c = g(d)$
- T_{PA} – 0, 1, +, *, =
- T_N – prirodni brojevi, +, =
- T_Z – cijeli brojevi,+, -, >, =, množenje konstantom
- ...
- Nešto detaljnije o odlučljivosti pojedinih teorija može se pročitati ovdje: http://web.stanford.edu/class/cs156/0910/slides/coc_technion_3.pdf

Model (semantika) teorije i pokazivanje ispravnosti

- Model M neke teorije T je definiran kao:
  - **Domena S** – skup elemenata
  - **Interpretacija funkcija f** za svaki f iz skupa funkcijskih simbola s brojem argumenata n: $f^M: S^n \rightarrow S$
  - **Interpretacija predikata P** za svaki P iz skupa predikatnih simbola s brojem argumenata n: $P^M \subseteq S^n$
  - **Pridruživanje** $x^M \in S$ za svaku varijablu x iz skupa varijabli

- M je model za formulu φ ako ju model evaluira u istinitu za dane interpretacije na domeni S
- M je model za cijelu teoriju T ako su sve formule iz T istinite za M (ako sve formule imaju model M)

Model (semantika) teorije i pokazivanje ispravnosti

- Formula $\varphi(x)$ je **zadovoljiva** s obzirom na teoriju T ako postoji barem jedan **model** M za T koji formulu $\varphi(x)$ evaluira u istinu.
  - Notacija: $M \vDash_T \varphi(x)$
- Formula $\varphi(x)$ je **valjana (ispravna, tautologija)** s obzirom na teoriju T ako se $\varphi(x)$ evaluira u istinu za svaki model M od T
- Alternativno: Formula $\varphi(x)$ je valjana ako je $\neg\varphi(x)$ nezadovoljiva
- Drugim riječima: ako je $\mathcal{A}$ skup aksioma koji opisuju neku teoriju T, **tada je formula $\varphi(x)$ valjana s obzirom na $\mathcal{A}$ ako je** $\mathcal{A} \cup \neg\varphi(x)$ nezadovoljiva
- Primijetiti: teorija T u ovoj definiciji može biti po volji složena teorija
- Korak prema formalnoj verifikaciji programa:
  - Da bismo provjerili je li formula `pre → program → post` ispravna, tu formulu trebamo negirati i pozvati SMT-rješavač koji treba pokazati da je ta negacija nezadovoljiva (UNSAT)

Primjer – model za formulu iz T_E

- Provjerimo je li tvrdnja:

  $\varphi \equiv a = b \wedge b = c \rightarrow g(f(a), b) = g(f(c), a)$ valjana u T_E

- Neka je M neki model od T_E. M **ne bi bio model** od φ jedino ako je $M \vDash_T$ $a = b \wedge b = c$ i $M \nvDash g(f(a), b) = g(f(c), a)$ (zbog implikacije $\rightarrow$)

- $M \vDash \forall x, y, z \,.\, x = y \wedge y = z \wedge x = z$ *(tranzitivnost)*

  $M \vDash a = c$

- $M \vDash \forall x, y.\ x = y \rightarrow f(x) = f(y)$ *(kong. funkcija)*

  $M \vDash f(a) = f(c)$

- $M \vDash a = b \wedge b = c$ *(elim. konjunkcije)*

  $M \vDash a = b$

Primjer – model za formulu iz T_E

- $M \vDash \forall x, y \,.\, x = y \wedge y = x$ *(komutativnost)*
  $M \vDash b = a$

- $M \vDash \forall x,y,u,v.\ x{=}y \wedge u{=}v \rightarrow g(x,u) = g(y,v)$ *(kongruentnost funkcija)*

- $M \vDash f(a) = f(c)$
- $M \vDash b = a$
- $M \vDash g(f(a), b) = g(f(c), a)$

- Došli smo do kontradikcije s pretpostavkom kada *M* ne bi bio model
- Zaključujemo: *M* je model od φ

Primjer – Peano aritmetika T_{PA}

- **Potpis**: konstante **0**, **1**, funkcije **+** i · te jednakost **=**
- Definirani su aksiomi za jednakost: refleksivnost, simetričnost, tranzitivnost, funkcije + i ·
- Dodatno, definirani su **aksiomi**:
  1. $\forall x.\ \neg(x + 1 = 0)$ (nula)
  2. $\forall x,y.\ x + 1 = y + 1 \rightarrow x = y$ (sljedbenik)
  3. $F[0] \wedge (\forall x. F[x] \rightarrow F[x+1]) \rightarrow \forall x. F[x]$ (indukcija)
  4. $\forall x.\ x + 0 = x$ (plus nula)
  5. $\forall x,y.\ x + (y + 1) = (x + y) + 1$ (plus sljedbenik)
  6. $\forall x.\ x \cdot 0 = 0$ (puta nula)
  7. $\forall x,y.\ x \cdot (y + 1) = x \cdot y + x$ (puta sljedbenik)

Primjer – Peano aritmetika T_{PA}

- Primjer: $3x + 5 = 2y$ može se zapisati u T_{PA} kao:

  $x + x + x + 1 + 1 + 1 + 1 + 1 = y + y$

  Napomena: mogli smo zapisati i

  $3x + 5 \geq 2y$ i to kao $\exists z.\ 3x + 5 = 2y + z$

- **T_{PA}** je **neodlučljiv** – s kvantifikacijom (Gödel, Tarski, Church, Turing, 1930-te) ili bez kvantifikacije varijable (Matiyasevich, 1970.),
- Fragment teorije **T_{PA} bez multiplikacije ($\cdot$) je odlučljiv**

Primjer – Teorija polja T_A

- **Potpis**: funkcija ***read***; funkcija ***write***; jednakost **=** (samo za elemente polja, ne cijela polja)
  - Funkcija *read(a, i)* je binarna funkcija koja čita iz polja *a* na indeksu *i*
  - Funkcija *write(a, i, v)* je ternarna funkcija: zapisuje vrijednost *v* na indeksu *i* polja *a*
- **Aksiomi:**
- $\forall a,i,j.\ i = j \rightarrow read(a, i) = read(a, j)$ (kongruentnost polja)
- $\forall a,v,i,j.\ i = j \rightarrow read(write(a, i, v), j) = v$ (read –write 1)
- $\forall a,v,i,j.\ i \neq j \rightarrow read(write(a, i, v), j) = read(a, j)$ (read –write 2)
- `a[i] = v` iz programa se prevodi u `a = write(a, i, v)`
- Loša vijest: T_A je **neodlučljiva**
- Dobra vijest: Fragment teorije T_A bez kvantifikatora je **odlučljiv**

Kombiniranje teorija

- Neki problemi zahtijevaju kombiniranje više teorija
- Problem kombiniranja više teorija je težak
  - Je li kombinacija dviju odlučljivih teorija opet odlučljiva teorija?
  - Što napraviti kada u programu imamo ograničenja koja kombiniraju više teorija?
- **Općeg rješenja nema**!
- Postoje korisni posebni slučajevi rješenja

Primjer: algoritam binarne pretrage treba kombinaciju teorija **aritmetike cijelih brojeva T_Z** i **polja T_A (*arrays*) –** rješivo za suvremene SMT-rješavače

```
int binary_search(
    int[] arr, int low, int high, int key) {
        assert (low > high || 0 <= low < high);
        while (low <= high) {
            //Find middle value
            int mid = (low + high)/2;
            assert (0 <= mid < high);
            int val = arr[mid];
            //Refine range
            if (key == val) return mid;
            if (val > key) low = mid+1;
            else high = mid-1;
        }
        return -1;
}
```

Kombiniranje teorija

- Kako pokazati da je formula:

$$1 \leq x \;\wedge\; x \leq 2 \;\wedge\; f(x) \neq f(1) \;\wedge\; f(x) \neq f(2)$$

ispravna u teoriji $(T_E \cup T_Z)$?

- Pokazuje se da ako postoje teorije T_1 i T_2 takve da im je presjek potpisa jednak {=}, onda zajednička teorija $(T_1 \cup T_2)$ ima potpis jednak uniji pojedinačnih potpisa i aksiome jednake uniji pojedinačnih aksioma.
- Nelson i Oppen su pokazali da ako je T_1 bez kvantifikatora odlučljiva, T_2 bez kvantifikatora odlučljiva i određeni jednostavni tehnički zahtjevi su ispunjeni, onda je i $(T_1 \cup T_2)$ bez kvantifikatora odlučljiva teorija

- Detalji: https://web.stanford.edu/class/cs357/lecture11.pdf

PRIMJER PRIMJENE I VEZA SA SAT-RJEŠAVAČIMA

Primjer: SMT-rješavač za raspoređivanje poslova koristeći aritmetiku razlike

- **Klasični problem raspoređivanja poslova** (engl. *job-scheduling*):
  - *n* poslova
  - Svaki posao sastoji se od *m* zadataka, svaki sa svojim trajanjem, koji se moraju slijedno izvoditi na *m* strojeva
  - Zadatak ne može biti prekinut jednom kada počne
- Ograničenja (2 vrste):
  - Prethođenje: zadan je redoslijed zadataka unutar jednog posla
  - Resursi: dva različita zadatka koji zahtijevaju isti stroj ne mogu biti pokrenuti istovremeno
- Zadano je najveće vrijeme *max* dokad svi poslovi moraju završiti

- **Zadatak:** na temelju zadanih trajanja zadataka svih poslova, pronaći raspored (ako postoji), takav da je konačno vrijeme izvršavanja svih zadataka svih poslova manje ili jednako *max*

Problem raspoređivanja poslova

$d_{i,j}$	Machine 1	Machine 2
Job 1	2	1
Job 2	3	1
Job 3	2	3

max $= 8$

Solution

$t_{1,1} = 5, \quad t_{1,2} = 7,$
$t_{2,1} = 2, \quad t_{2,2} = 6,$
$t_{3,1} = 0, \quad t_{3,2} = 3$

Encoding

$$\begin{aligned}
&(t_{1,1} \geq 0) \wedge (t_{1,2} \geq t_{1,1} + 2) \wedge (t_{1,2} + 1 \leq 8) \wedge \\
&(t_{2,1} \geq 0) \wedge (t_{2,2} \geq t_{2,1} + 3) \wedge (t_{2,2} + 1 \leq 8) \wedge \\
&(t_{3,1} \geq 0) \wedge (t_{3,2} \geq t_{3,1} + 2) \wedge (t_{3,2} + 3 \leq 8) \wedge \\
&((t_{1,1} \geq t_{2,1} + 3) \vee (t_{2,1} \geq t_{1,1} + 2)) \wedge \\
&((t_{1,1} \geq t_{3,1} + 2) \vee (t_{3,1} \geq t_{1,1} + 2)) \wedge \\
&((t_{2,1} \geq t_{3,1} + 2) \vee (t_{3,1} \geq t_{2,1} + 3)) \wedge \\
&((t_{1,2} \geq t_{2,2} + 1) \vee (t_{2,2} \geq t_{1,2} + 1)) \wedge \\
&((t_{1,2} \geq t_{3,2} + 3) \vee (t_{3,2} \geq t_{1,2} + 1)) \wedge \\
&((t_{2,2} \geq t_{3,2} + 3) \vee (t_{3,2} \geq t_{2,2} + 1))
\end{aligned}$$

$d_{i,j}$ = trajanje *j*. zadatka *i*. posla
$t_{i,j}$ = početak *j*. zadatka *i*. posla

Kodiranje početnog i završnog vremena:
$t_{i,1} \geq 0$: početno vrijeme 1. zadatka za *i*. posao
$t_{i,m} + d_{i,m} \leq max$: završno vrijeme zadnjeg *m*. zadatka najviše jednako *max*

Kodiranje prethođenja:
$t_{i,j+1} \geq t_{i,j} + d_{i,j}$: novi zadatak počinje kad prethodni završi

Kodiranje resursa:
$(t_{i,j} \geq t_{i',j} + d_{i',j}) \vee (t_{i',j} \geq t_{i,j} + d_{i,j})$: dva zadatka *j* od različitih poslova *i* te *i'* se ne izvode na istom stroju u isto vrijeme

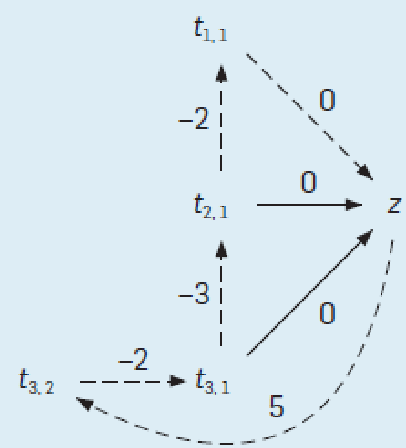
Teorija aritmetike razlike

- **Aritmetika razlike** (engl. *difference arithmetic*) podvrsta je **linearne aritmetike** T_Z **(linearna ograničenja nad cijelim brojevima)** u kojoj se dozvoljavaju samo formule oblika:

  $t - s \leq c$, gdje su t i s varijable, a c je numerička konstanta
- Ako su varijable u našem problemu zadane u obliku $s \leq c$ ili $c \leq s$ može ih se svesti na traženi oblik aritmetike razlike koristeći "nultu" varijablu $z = 0$, čime se dobiva: $s - z \leq c$ ili $z - s \leq -c$
- Aritmetiku razlike može se iskoristiti za riješiti klasični problem raspoređivanja poslova, budući da se sva ograničenja mogu prikazati u tom obliku
- Iz ograničenja u tom obliku oblikuje se **težinski usmjereni graf**

Težinski usmjereni graf – primjer

$$
\begin{array}{llll}
z & - \; t_{1,1} & \leq & 0 \\
z & - \; t_{2,1} & \leq & 0 \\
z & - \; t_{3,1} & \leq & 0 \\
t_{3,2} & - \; z & \leq & 5 \\
t_{3,1} & - \; t_{3,2} & \leq & -2 \\
t_{2,1} & - \; t_{3,1} & \leq & -3 \\
t_{1,1} & - \; t_{2,1} & \leq & -2
\end{array}
$$



- Svaka varijabla je čvor, a nejednakost t - $s \leq c$ odgovara usmjerenom bridu od varijable s u varijablu t s težinom c
- Skup varijabli u aritmetici razlike može se učinkovito pretražiti za zadovoljivost tako da se traže **negativni ciklusi** u težinskom usmjerenom grafu.
- **Negativni ciklus je put kroz težinski usmjereni graf od čvora *x* nazad do čvora *x* na kojem je ukupan zbroj težina negativan**
- U ovom konkretnom primjeru prikazan je samo dio ukupnog težinskog usmjerenog grafa, gdje negativan ciklus čvora $t_{3,2}$ počinje putom -2 prema $t_{3,1}$ i nastavlja se po slijedu crtkanih linija; sve nazad to $t_{3,2}$, što odgovara rasporedu poslova posao1/zadatak1, posao2/zadatak1, posao3/zadatak1, posao3/zadatak2 koji **ne može završiti u *max* vremenu**

Pokazivanje zadovoljivosti – pristup “lazy offline”

- Problem ograničenja u aritmetici razlike preslikava se u SAT-problem tako da se uvedu **nove Booleove varijable**, npr. umjesto $\neg(a \geq 3) \wedge (a \geq 3 \vee a \geq 5)$ Booleova formula bi glasila $\neg p1 \wedge (p1 \vee p2)$.
- **Ako SAT-oblik pokaže nezadovoljivost, onda je i problem ograničenja nezadovoljiv, a ako pokaže zadovoljivost, onda se dalje iskoristi SMT-rješavač da se provjeri model od SAT-a**
  - Npr. ($p1$ = false, $p2$ = true) je model od SAT-a za gornji primjer, a kad se taj model prevede u SMT: $(\neg(a \geq 3), a \geq 5)$, to je nezadovoljivo u teoriji linearne aritmetike.

$p1$ $p2$

Pokazivanje zadovoljivosti – pristup “lazy offline”

- Budući da je SMT bio nezadovoljiv, dalje se negira prethodna formula: $(a \geq 3 \vee \neg(a \geq 5))$ čime ovo postaje valjana SMT formula i čija je apstrakcija u SAT-obliku $p1 \vee \neg p2$.
- Ovu izvedenu klauzulu zovemo “**lema teorije**”.
- Budući da je ova lema SMT teorije valjana formula, može se dodati originalnoj formuli i dobiti $\neg p1 \wedge (p1 \vee p2) \wedge (p1 \vee \neg p2)$
- Ponovno se pokrene SAT-rješavač koji sada **ne pronalazi model**
- Ovime smo dokazali da je izvorna formula:

  $\neg(a \geq 3) \wedge (a \geq 3 \vee a \geq 5)$ nezadovoljiva.

Pokazivanje zadovoljivosti – pristup "lazy offline"

- U praksi, u nizu koraka stvara se puno lema teorije dok se ne dođe do konačnog odgovora o modelu koji može biti:
  - **SAT i SMT kažu da postoji model ili**
  - **SAT kaže da nema modela**
- **Rješenje se uvijek pronalazi jer imamo konačni broj varijabli i odlučljivu teoriju**
- U našem primjeru, negacija nezadovoljivog negativnog ciklusa u težinskom usmjerenom grafu bila bi valjana:

  $\neg(t_{3,1} - t_{3,2} \leq -2) \vee \neg(t_{2,1} - t_{3,1} \leq -3) \vee \neg(t_{1,1} - t_{2,1} \leq -2) \vee \neg(z - t_{1,1} \leq 0) \vee \neg(t_{3,2} - z \leq 5)$ i činila bi lemu teorije koja bi se onda dalje dokazivala...

Najuspješniji SMT-rješavači

- Svi navedeni podržavaju rješavanje većine odlučljivih teorija:
  - **Z3** (Microsoft) (2008.+) – *https://github.com/Z3Prover/z3*
  - **Yices 2** (SRI internation, NSF, NASA, DARPA) (2009.+) – *http://yices.csl.sri.com/*
  - CVC4 (Stanford University) (2013.+) – *http://cvc4.cs.stanford.edu/web/*
  - MathSAT 5 (University of Trento) (2013.+) – *http://mathsat.fbk.eu/*
  - Boolector (Johannes Kepler University, Austria) (2009.+) – *https://boolector.github.io/*
- Ulazni jezik: SMT-lib - https://smtlib.cs.uiowa.edu/
- Godišnja natjecanja SMT-rješavača: *https://smt-comp.github.io*

UVOD U SIMBOLIČKO IZVRŠAVANJE PROGRAMA

Cilj, zadatak i problemi...

- **Cilj:** automatsko pronalaženje kvarova u programu generiranjem ispitnih slučajeva koji ih prokazuju
- **Zadatak**: potrebno je proći kroz sve putove programa (idealno), ili kroz što veći broj putova u ograničenom vremenu (realno)
- Simboličko izvršavanje programa poznato je već skoro 50 godina (1975.+), ali se tek odnedavno uspješno ostvaruje, zbog snažnog razvoja SAT-rješavača i rješavača ograničenja (engl. *constraint solver*)
- Problemi koje simboličko izvršavanje želi riješiti i na kojima se radi:
  - Grananja pri izvršavanju uzrokuju eksponencijalni porast broja putova
  - Petlje generiraju vrlo veliki broj putova
  - Varijable programa mogu poprimiti veliki raspon vrijednosti i razne vrste ograničenja
  - Istovremeno izvršavanje je problematično za analizu i ispitivanje
  - Komunikacija s vanjskim knjižnicama, upravljanje memorijom...

Značajke simboličkog izvršavanja

- Ključne značajke simboličkog izvršavanja:
  1. **Za ulazne vrijednosti programa koriste se simboličke vrijednosti umjesto konkretnih vrijednosti**
  2. **Varijable programa prikazuju se kao simbolički izrazi nad simboličkim vrijednostima**
  3. **Pri ispitivanju programa, simboličko izvršavanje se koristi za generiranje konkretnih ulaznih podataka za svaki ostvarivi put kroz program**
- Tijekom izvođenja programa dodaju se nova ograničenja nad varijablama

Značajke simboličkog izvršavanja

- Simboličko izvršavanje programa održava u svakom trenutku:
  - σ - simboličko stanje (engl. *symbolic state, symbolic store*)
    - Preslikava dotad posjećene varijable programa u simboličke izraze
  - SPC - simboličko ograničenje puta (engl. *symbolic path constraint, symbolic path condition*)
    - Služi za pamćenje svih dotad posjećenih **grananja** u programu
    - Logička formula u odlučljivom obliku logike prvoga reda bez kvantifikatora
    - Razrješava se da bi se našle konkretne vrijednosti na kraju ostvarivog puta

Primjer simboličkog izvršavanja

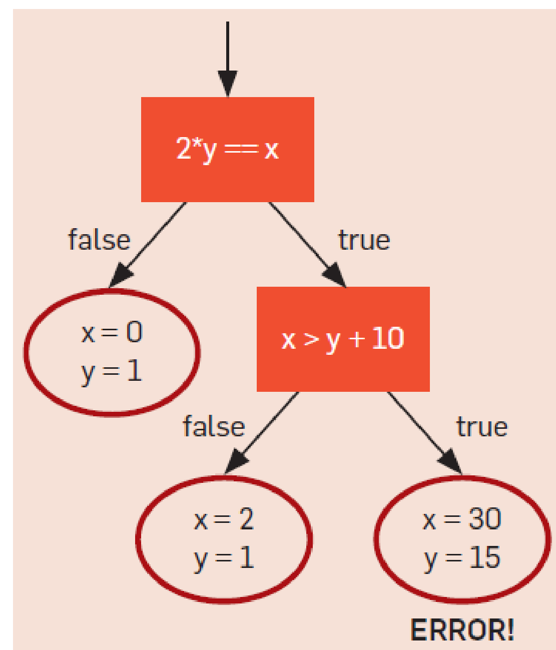
Program:

```
int twice(int v) {
  return 2 * v;
}

void testme(int x, int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  testme(x,y);
}
```

- Zadatak je utvrditi koji ulazni podatci dovode izvođenje programa do retka “ERROR”
- U praksi, “ERROR” može biti bilo koja vrsta pogreške
- Budući da ne znamo *a priori* gdje se pogreška nalazi, trebamo ispitati sve putove
- Postoje tri moguća puta kroz program, primjer stabla izvršavanja:



Primjer simboličkog izvršavanja

Program:

```
int twice(int v) {
  return 2 * v;
}

void testme(int x, int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  testme(x,y);
}
```

- Kako se odvija simboličko izvršavanje ovog programa?

1. Funkcije `read` dobavljaju ulazni podatak odnekud (npr. iz komandne linije), ne znamo njihovu vrijednost
2. Postavljaju se varijable *x* i *y* na nove simboličke vrijednosti

- Početno stanje simboličkog izvršavanja prije funkcije `testme(x,y)` izgleda ovako:

$\sigma : x \rightarrow x0$
$\quad\;\; y \rightarrow y0$

SPC = true
(zasad još nema posjećenih grananja)

Primjer simboličkog izvršavanja

Program:

```
int twice(int v) {
  return 2 * v;
}

void testme(int x, int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  testme(x,y);
}
```

3. Izvršava se funkcija `twice`, dobiva se novo stanje:

σ : x $\rightarrow$ x0 **SPC** = true
y $\rightarrow$ y0
z $\rightarrow$ 2*y0

4. Izvršavanje se dijeli na 2 puta (dupliciraju se stanja programa):

za (`x == z`):

σ : x $\rightarrow$ x0 **SPC** : x0 = 2*y0
y $\rightarrow$ y0
z $\rightarrow$ 2*y0

za (`x` $\neq$ `z`):

σ : x $\rightarrow$ x0 **SPC** : x0 $\neq$ 2*y0
y $\rightarrow$ y0
z $\rightarrow$ 2*y0

Moglo bi se stati s istraživanjem pojedinog puta ako bi se znalo da je SPC nezadovoljiv. U ovom slučaju, oba SPC-a su zadovoljiva pa se nastavlja.

Primjer simboličkog izvršavanja

Program:

```
int twice(int v) {
  return 2 * v;
}

void testme(int x, int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
       ERROR;
  }
}

int main() {
  x = read();
  y = read();
  testme(x,y);
}
```

Neka se dalje istražuje put za `(x == z)` :

5. Dijeli se dalje na 2 puta (ponovno se duplicira stanje programa:

Ovim putem smo došli do retka ERROR

za `(x > y + 10)`:

σ : $x \rightarrow x0$, $y \rightarrow y0$, $z \rightarrow 2*y0$

SPC : $x0 = 2*y0 \wedge x0 > y0 + 10$

za `(x ≤ y + 10)`:

σ : $x \rightarrow x0$, $y \rightarrow y0$, $z \rightarrow 2*y0$

SPC : $x0 = 2*y0 \wedge x0 \leq y0 + 10$

Obrada ograničenja

- U svakom trenutku izvođenja programa, može se pokrenuti **rješavač ograničenja** (danas najčešće SMT-rješavač) koji će pronaći pridruživanje varijablama koje će biti model od SPC
- Pokretanje rješavača obično se provodi:
  - **na mjestima grananja**, kako bi se spriječilo daljnje istraživanje putova ako su ograničenja za taj put nezadovoljiva ili
  - **na mjestu gdje je došlo do pogreške**, kako bi se utvrdio **primjer konkretnih ulaznih vrijednosti** koji je doveo do pogreške
    - Npr. za (`x > y + 10`) došli smo do retka ERROR i razrješava se ograničenje `x0 = 2*y0 ∧ x0 > y0 + 10`
- Tako se pronalaze pogreške u složenim programima koje klasični postupci ispitivanja ne mogu jednostavno pronaći

KONKRETNO/SIMBOLIČKO IZVRŠAVANJE

Problem simboličkog izvršavanja

- **Ključni problem klasičnog simboličkog izvršavanja: ispitni slučaj ne može se generirati ako SMT-rješavač ne može (učinkovito) razriješiti ograničenje**

- Primjer: za nelinearna ograničenja koja se rješavaju u vidu T_{PA} (koja uključuje multiplikaciju), **nemamo garanciju** da će SMT-rješavač uspjeti pronaći model ili opovrgnuti SPC
  - Slično vrijedi i za neke kombinacije teorija, a problem je i komunikacija s vanjskim knjižnicama koje koriste funkcije za koje ne znamo kako se ponašaju

Rješenje za simboličko izvršavanje

- **Kombinirati klasično konkretno izvršavanje i simboličko izvršavanje, tzv. konkoličko izvršavanje (engl. *concolic execution, concolic testing*)**
- Danas se najčešće koristi pristup **dinamičkog simboličkog izvršavanja** (engl. *dynamic symbolic execution*, DSE)
  - Konkretno izvršavanje upravlja simboličkim izvršavanjem tako što se **program izvodi s konkretnim vrijednostima varijabli, ali se uz to pamte i simboličke vrijednosti** (program ima konkretno i simboličko stanje)
  - Nema garancije da će se istražiti svi putovi
- Postoje i alternative, npr. izvršavanjem generirano ispitivanje (engl. *execution generated testing*, EGT), selektivno simboličko izvršavanje (engl. *selective symbolic execution*, S^2E), itd.

DSE – primjer izvršavanja

Program:

```
int twice(int v) {
  return (v * v) % 50;
}

void testme(int x, int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
       ERROR;
  }
}

int main() {
  x = read();
  y = read();
  testme(x,y);
}
```

- Izmijenjeni primjer sadrži nelinearno ograničenje

za (`x == z`):

σ : $x \rightarrow x0$

$y \rightarrow y0$

$z \rightarrow (y0*y0)\%50$

SPC : $x0 = (y0*y0)\%50$

Problem!

Primjer DSE-izvršavanja „problematičnog” programa

Program:

```
int twice(int v) {
  return (v * v) % 50;
}

void testme(int x, int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  testme(x,y);
}
```

- Funkcija `read()` dobavlja konkretne podatke, npr. x = 22, y = 7:

$\sigma_k : x \rightarrow 22$, $y \rightarrow 7$ **SPC** = true (zasad još nema posjećenih grananja)

$\sigma : x \rightarrow x0$, $y \rightarrow y0$ **SPC** = true (zasad još nema posjećenih grananja)

$\sigma_k : x \rightarrow 22$, $y \rightarrow 7$, $z \rightarrow 49$ **SPC** = true (zasad još nema posjećenih grananja)

$\sigma : x \rightarrow x0$, $y \rightarrow y0$, $z \rightarrow (y0*y0)\%50$ **SPC** = true (zasad još nema posjećenih grananja)

- Ide se granom (`x` $\neq$ `z`):

$\sigma_k : x \rightarrow 22$, $y \rightarrow 7$, $z \rightarrow 49$ **SPC :** $x0 \neq (y0*y0)\%50$

$\sigma : x \rightarrow x0$, $y \rightarrow y0$, $z \rightarrow (y0*y0)\%50$ **SPC :** $x0 \neq (y0*y0)\%50$

Primjer DSE-izvršavanja „problematičnog” programa

Program:

```
int twice(int v) {
  return (v * v) % 50;
}

void testme(int x, int y){
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  testme(x,y);
}
```

$\sigma_k : x \rightarrow 22$
$\quad y \rightarrow 7$
$\quad z \rightarrow 49$

SPC : $x0 \neq (y0*y0)\%50$

$\sigma : x \rightarrow x0$
$\quad y \rightarrow y0$
$\quad z \rightarrow (y0*y0)\%50$

SPC : $x0 \neq (y0*y0)\%50$

- Ovdje imamo nelinearno ograničenje `x0 ≠ (y0*y0)%50`. Ako želimo istražiti drugu granu, dakle za (`x == z`), negiramo ograničenje, no opet dobivamo nelinearno ograničenje `x0 = (y0*y0)%50`
- Ovdje DSE-izvršavanje pojednostavi ograničenje ubacivanjem konkretne vrijednosti y0 = 7, čime se rješavanjem SPC dobiva x0 = 49
- Zatim ponovno pokreće program s ulazom: `x → 49, y → 7`
- Sad zaista dolazimo do retka "ERROR"
- Međutim, ako iskoristimo DSE-izvršavanje, onda u ovom slučaju ne istražimo granu (`x ≤ y + 10`)

Problem eksplozije broja putova

- Kako bi se istražili i alternativni putovi, potrebno je svaki put **negirati SPC** do kojeg se došlo u ranijem istraživanju
- Istraživanje samo mogućih putova, pokazalo se:
  - Manje od 20% grananja imaju oba puta ostvariva*
  - Manje od 42% naredbi ovise o simboličkom *inputu**
- Heuristike pretraživanja putova (prostora stanja)
  - U dubinu – naprije se istražuju najdublja grananja
  - Slučajni put – na svakom grananju gdje su oba puta ostvariva, odaberi slučajno put koji treba dalje istražiti - začudo, pokazuje se izvrsnom heuristikom
  - Korištenje statičkog grafa kontrolnog toka (engl. *control flow graph*) programa, da se usmjeri istraživanje puta najbližeg dotad nepokrivenoj naredbi

* Cadar, C., Ganesh, V., Pawlowski, P., Dill, D. and Engler, D. EXE: Automatically generating inputs of death. In *Proceedings of CCS'06, (Oct–Nov 2006). An extended* version appeared in ACM TISSEC 12, 2 (2008).

Alati za simboličku verifikaciju

- DART (2005.+)
  - Izvorno konkoličko ispitivanje: slučajno ispitivanje + generiranje ispitnih slučajeva, provjera modela -> jezik C
- CUTE (*A Concolic Unit Testing Engine*) i jCUTE (*CUTE for Java*) (2006.+)
  - Nadogradnja DART-a za rad s višedretvenošću
  - *http://osl.cs.illinois.edu/software/jcute/*
- EXE i KLEE (2008.+)
  - Korite izvršavanjem generirano ispitivanje. veliki broj istovremenih stanja, ispitivanje programa u raznim programskim jezicima nakon pretvorbe u LLVM međujezik
  - *https://klee.github.io/*
- S^2E (2011.+)
  - Virtualni stroj sa selektivnim simboličkim izvršavanjem i modularnim analizatorima puta
  - *http://s2e.epfl.ch/*
- Java PathFinder, paket "symbolic" (2008.+):
  - *https://github.com/SymbolicPathFinder/jpf-symbc*

Materijali pripremljeni na temelju

1. L. de Moura, N. Bjørner, "Satisfiability Modulo Theories: Introduction and Applications," Communications of the ACM vol. 54, no. 9, pp.69-77, Sep. 2011
2. R. Piskač, Software Aided Verification, Lectures 8-15, Yale University, 2020
3. C. Cadar, K. Sen, "Symbolic Execution for Software Testing: Three Decades Later," Communications of the ACM vol. 56, no. 2, pp.82-90, Feb. 2013.
4. R. Baldoni et al., "A Survey of Symbolic Execution Techniques", ACM Computing Surveys, vol. 51, issue 3, p. 50, pp 1–39, 2019.
5. M. Vechev, "Software Architecture and Engineering: Part II," Software Reliability Lab, ETH Zurich, 2015.
6. D. Paqué, "From Symbolic Execution to Concolic Testing," Institute of Theoretical Computer Science, Carl-Friedrich-Gauß-Fakultät , Technische Universität Braunschweig, 2014.